AD________________

Award Number: DAMD17-02-2-0048

TITLE: Monitoring and Mining Data Streams

PRINCIPAL INVESTIGATOR: Stan B. Zdonik , Ph.D.

CONTRACTING ORGANIZATION: Brown University
Providence, RI 02912

REPORT DATE: October 2003

TYPE OF REPORT: Annual

PREPARED FOR: U.S. Army Medical Research and Materiel Command
Fort Detrick, Maryland 21702-5012

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** October 2003	**3. REPORT TYPE AND DATES COVERED** Annual (15 Sep 2002 - 14 Sep 2003)

4. TITLE AND SUBTITLE Monitoring and Mining Data Streams	**5. FUNDING NUMBERS** DAMD17-02-2-0048
6. AUTHOR(S) Stan B. Zdonik, Ph.D.	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Brown University Providence, RI 02912 E-Mail: sbz@cs.brown.edu	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) U.S. Army Medical Research and Materiel Command Fort Detrick, Maryland 21702-5012	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for Public Release; Distribution Unlimited	**12b. DISTRIBUTION CODE**

13. ABSTRACT (Maximum 200 Words)

The work of the first year of this effort was to support the implementation of the prototype for the Aurora stream processing engine and to investigate applications for this technology. An initial version of Aurora was completed, and we worked with William Vanderschallie on an Army application that involved water supply threat-level detection. The rest of this report discusses these two activities.

14. SUBJECT TERMS	**15. NUMBER OF PAGES** 14
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** Unlimited

Table of Contents

Annual Report

DEPARTMENT OF THE ARMY
US ARMY MEDICAL RESEARCH AND MATERIEL COMMAND
504 SCOTT STREET
FORT DETRICK, MD 21702-5012

Award Number: DAMD17-02-2-0048

PI's: Stan Zdonik and Ugur Cetintemel

Brown University
Dept. of Computer Science
P.O. Box 1910
Providence, RI 02912

Abstract

The work of the first year of this effort was to support the implementation of the prototype for the Aurora stream processing engine and to investigate applications for this technology. An initial version of Aurora was completed, and we worked with William Vanderschallie on an Army application that involved water supply threat-level detection. The rest of this report discusses these two activies.

Overview of Aurora

The Aurora project represents a new category of data management infrastructure known as Data Stream Management Systems (DSMS). DSMS's are a promising new approach to information processing that provides continual monitoring and analysis of data streams. Applications range from financial (e.g., scan stock trade activity for fraud) to military (e.g., alert when enemy units are inbound) to environmental (e.g., sound an alarm when bad water quality is detected).

The main hypothesis examined by this project is that there are compelling reasons to implement stream-processing applications on top of a DSMS, rather than with alternative technologies such as procedural programming or with a relational database. A DSMS-based stream-processing application can reap the following benefits:

- **Speed.** For a given commodity computer, a DSMS-based application should be able to process input data faster than a traditionally designed application can. (An example of a traditionally designed system would be one that uses a relational database to store intermediate data.)
- **Resilience to unpredictable stream behavior.** It's difficult to write software that makes good use of data that are delayed, lost, or arrive out of order. An application that's handles these data problems well should be easier to write with a DSMS than with traditional technology.
- **Simplified / quick programming.** Writing software to perform complex manipulation of data streams can be difficult. It should be faster to write a powerful stream-monitoring application with a DSMS than with traditional technology (e.g., procedural programming).

We validated these hypotheses by writing DSMS-based applications for various users, described below. Based on those experiences, we concluded that these hypotheses are correct: writing stream-processing applications on top of a DSMS has many compelling benefits.

Core Software

The Aurora DSMS (hereafter called *Aurora*) is a client-server system that runs on the Linux operating system. The server component performs the actual data processing. The client component, which may run on different computers than the server, sends streams of data into the server and receives any alerts / results produced by the server.

Aurora provides a GUI to let users visually describe the data processing that *Aurora* is to perform, in a language named *SQuAl*. With this GUI, even relatively novice users can construct or alter sophisticated data stream monitoring applications.

SQuAl

SQuAl is *Aurora*'s visual language that describes the flow of data between data-manipulation operators. A SQuAl program looks much like a flow-chart. The boxes represent data-manipulation operators, and the arrows connecting the boxes show how the data flow from one operator to the next.

SQuAl's operators are specially designed to intelligently handle delayed data or missing data. For example, a Sort operator will take slightly out-of-order data as its input, and emit reordered data that's properly sorted. This finesse for handling delayed / missing data makes SQuAl-based applications unusually robust in hostile environments, such as when battlefield sensors are temporarily or permanently disabled.

Performance

Aurora's processing throughput is extremely high compared to that of a relational database. *Aurora* has been shown to process nearly 200,000 data tuples per second for a non-trivial application on a commodity PC with a 2.8 GHz microprocessor.

Applications Developed

The following *Aurora*-based applications were created to verify that *Aurora* can be used to solve real-world problems.

Some of these applications provided feedback that led to improvements in the SQuAl language or in the implementation of the *Aurora* software. Ultimately *Aurora* and SQuAl proved to be well suited to these applications.

Financial Services Application

Financial service organizations purchase stock ticker feeds from multiple providers and need to switch in real-time between these feeds if they experience too many problems. We worked with a major financial services company on developing an Aurora application that detects feed problems and triggers the switch in real time.

In this application we were able to capitalize *Aurora*'s robust handling of irregularly timed input data. Additionally, one PC running the Aurora-based application significantly outperformed that company's alternative implementation running on a high-end Sun server.

Linear Road Benchmark

Linear Road is a benchmark for stream processing engines such as *Aurora*. This benchmark simulates an urban highway system that uses "variable tolling" (also known as "congestion pricing"), where tolls are determined according to such dynamic factors as congestion, accident proximity, and travel frequency. As a benchmark, Linear Road specifies fixed input data schemas and workloads, a suite of continuous and historical queries that must be supported, and performance (query and transaction response time) requirements.

An early *Aurora*-based implementation of this benchmark supporting one expressway was demonstrated at SIGMOD 2003. We're presently developing an alternative implementation of this benchmark with a popular relational database, to compare its performance to that of our *Aurora*-based implementation.

Environmental Monitoring

We have also worked with a military medical research laboratory (William Vanderschallie) on an application that involves monitoring toxins in the water. This application is fed streams of data regarding fish behavior (e.g., breathing rate) and water quality (temperature, pH, oxygenation, and conductivity). When the fish behave abnormally, an alarm is sounded.

Input data streams were supplied by the army lab as a text file. The single data file interleaved fish observations with water quality observations. The alarm message emitted by *Aurora* contains fields describing the fish behavior, and two different water quality reports: the water quality at the time the alarm occurred and the water quality from the last time the fish behaved normally. The water quality reports contain not only the simple measurements, but also the 1-/2-/4-hour sliding window deltas for those values.

During the development of the application, we observed that *Aurora*'s stream model proved very convenient for describing the required sliding-window calculations. For example, a single instance of an operator computed the 4-hour sliding-window deltas of water temperature.

Aurora's GUI for designing query networks also proved invaluable. It let us easily understand the processing that was to take place, even as our SQuAl program grew to over 50 operators.

The ease with which the processing flow could be experimentally reconfigured during development, while remaining comprehensible, was surprising. It appears that this was only possible both by SQuAl having both a well-suited operator set, and by having a GUI tool that let us visualize the stream processing.

Please see this document's appendix for a fuller description of this application.

Medusa: Distributed Stream Processing

Over the last year, we have worked closely with the Medusa project at MIT.

Medusa is a distributed stream-processing system that uses Aurora as its single-site processing engine. Medusa takes Aurora queries and distributes them across multiple computers. These computers can all be under the control of one entity or can be organized as a loosely coupled federation under the control of different autonomous *participants*.

A distributed stream-processing system such as Medusa offers several benefits:

1. It allows stream processing to be incrementally scaled over multiple nodes.
2. It enables high-availability because the processing nodes can monitor and take over for each other when failures occur.
3. It allows the composition of stream feeds from different participants to produce end-to-end services, and to take advantage from the distribution inherent in many stream processing applications (e.g., climate monitoring, financial analysis, etc.).
4. It allows participants to cope with load spikes without individually having to maintain and administer the computing, network, and storage resources required for peak operation. When organized as a loosely coupled federated system, load movements between participants based on predefined contracts can significantly improve performance.

Battalion Monitoring

We have worked closely with a major defense contractor on a battlefield monitoring application. In this application, an advanced aircraft gathers reconnaissance data and sends it to monitoring stations on the ground. This data includes positions and images of friendly and enemy units. At some point, the enemy units will cross a given demarcation line and move toward the friendly units thereby signaling an attack.

Commanders in the ground stations monitor this data for analysis and tactical decision making. Each ground station is interested in particular subsets of the data, each with differing priorities. In the real application, the limiting resource is the bandwidth between the aircraft and the ground. When an attack is initiated, the priorities for the data classes change. More data become critical, and the bandwidth likely saturates. In this case, selective dropping of data is allowed in order to service the more important classes.

For our purposes, we built a simplified version of this application to test our load shedding techniques. Instead of modeling bandwidth, we assume that the limited resource is the CPU. We introduce load shedding as a way to save cycles.

Quality-of-Service-based Load Shedding

We continue to investigate strategies of intelligently dropping non-critical data from the streams *Aurora* processes, during resource-critical moments. We're investigate ways of letting users express which data are considered most critical in these scenarios.

Scheduling

In data processing, there are often trade offs between latency and throughput. We continue to explore the effects of this phenomenon in stream processing, and for techniques to maximize both of these important qualities in a DSMS.

High Availability

We are also currently exploring the runtime overhead and recovery time trade offs between different approaches to achieve high-availability

(HA) in distributed stream processing, in the context of Medusa and *Aurora**. These approaches range from classical Tandem-style process-pairs to using upstream nodes in the processing flow as backup for their downstream neighbors. Different approaches also provide different recovery semantics where either: (1) some tuples are lost, (2) some tuples are re-processed, or (3) operations take-over precisely where the failure happened. An important HA goal for the future is handling network partitions in addition to individual node failures.

Papers and Conferences

- N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, M. Stonebraker. **Load Shedding in a Data Stream Manager**. In proceedings of the *29th International Conference on Very Large Data Bases* (*VLDB*'03), Berlin, Germany, September 2003 (to appear).
- Don Carney, Ugur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack and Michael Stonebraker. **Operator Scheduling in a Data Stream Environment**, In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*, September, 2003.
- D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S. Zdonik. **Aurora: A New Model and Architecture for Data Stream Management**. In *VLDB Journal*, August 2003 (to appear).
- D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, J. Hwang, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R.Yan, S. Zdonik. **Aurora: A Data Stream Management System (Demonstration)**. In proceedings of the *ACM SIGMOD International Conference on Management of* Data (*SIGMOD*'03), San Diego, CA, June 2003.
- D. Carney, U. Çetintemel, A. Rasin, S. Zdonik, M. Cherniack, M. Stonebraker. **Reducing Execution Overhead in a Data Stream Manager**. In proceedings of the *ACM Workshop on Management and Processing of Data Streams* (*MPDS*'03), San Diego, CA, June 2003.
- N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, M. Stonebraker. **Load Shedding on Data Streams**. In proceedings of the *ACM Workshop on Management and Processing of Data Streams* (*MPDS*'03), San Diego, CA, June 2003.
- D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, J. Hwang, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan and S. Zdonik, **Aurora: A Data Stream Management System (Demonstration)**, In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Diego, CA, June, 2003.
- S. Zdonik, M. Stonebraker, M. Cherniack, U. Çetintemel, M. Balazinska, and H. Balakrishnan. **The Aurora and Medusa Projects**. *Bulletin of the Technical Committee on Data Engineering, IEEE Computer Society*. March 2003 (*invited Paper*).
- M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, S. Zdonik. **Scalable Distributed Stream Processing.** In proceedings of the *First Biennial Conference on Innovative Database Systems* (*CIDR*'03), Asilomar, CA, January 2003.

Appendix: Environmental Monitoring Application

The Problem

Keeping public water supplies clean and safe is difficult. A tremendous number of different substances, both known and unknown, can enter the water supply and harm people and animals.

Unfortunately it's practically impossible to develop mechanized tests to detect the presence of all such *known* toxins. Devising tests to alert people to the presence of substances previously *not known* to be toxic, without sounding too many false alarms, is an even bigger challenge.

One solution to this problem is to expose test organisms to the water, and to continuously monitor their behavior. A well chosen organism will be sensitive to the same chemicals that are harmful to humans and our domestic animals. One strength of using organisms to monitor the water is that the presence of previously unanticipated / unknown toxins may still cause an alarm to sound. If standard laboratory equipment, rather than organisms, was used to monitor the water quality then there would be a smaller chance of noticing the presence of these unanticipated toxins.

This technique was previously practiced by having miners bring canaries into the mineshafts. If the canaries were overcome by poisonous gasses in the mines, then the miners knew it was time to head for the surface.

The motivation for using biological monitoring systems for our water supplies is therefore clear. For cost and efficiency reasons, however, we'd like to automate this process as much as possible. This data processing can be difficult to implement with traditional technologies, but is well suited to a DSMS such as *Aurora.*

Context

One U. S. Army laboratory has applied the biological monitoring technique to the water passing through a groundwater treatment plant. In their case, they used a set of fish swimming in that water as the "canaries".

Each of the fish swam in its own tube, through which the treated water flowed. Each fish's behavior was monitored using small sensors implanted on the fish that tracked how well the fish were breathing, and how much their overall bodies were moving. These measurements were provided by the sensor system every 15 minutes.

An alarm was to be sounded whenever a large fraction of those fish behaved abnormally. The presumption was that the fish were responding to unusual qualities in the water.

Water quality samples were also periodically taken using traditional automated laboratory equipment. The samples recorded the water's pH, conductivity, oxygenation level, and temperature. The reason for taking these samples was to give a person responding to a quality alarm additional information to determine what was wrong with the water. These measurements were provided by the sensor system once per hour.

When an alarm was sounded because enough fish were behaving abnormally, our application was to produce an alert message structured as follows:

- It must contain fields describing the current fish behavior, and two different water quality reports: the water quality at the time the alarm occurred and the water quality from the last time the fish behaved normally.

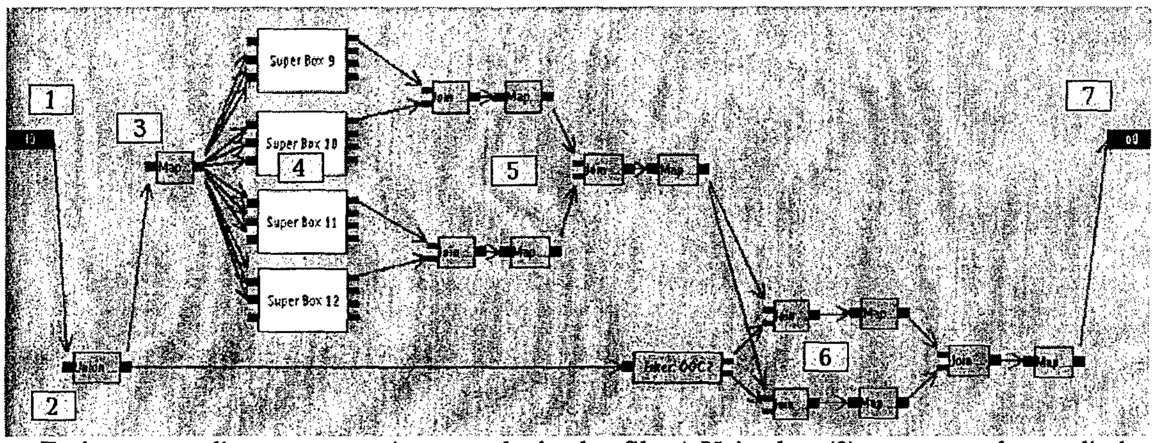

Figure 1: Aurora SQuAl program for environmental monitoring

- Each water quality report contains not only the simple measurements (pH, conductivity, oxygenation, and temperature), but also the 1-/2-/4-hour sliding window deltas for those values.

The Solution

Input data streams were supplied by the Army lab as a text file. The single data file interleaved fish observations with water quality observations.

We developed several pieces of software for this application:

- A C++ program for (a) reading the text file provided by the Army, and sending those sensor readings into *Aurora,* and (b) showing the water quality alarms emitted by *Aurora.*
- A SQuAl program that monitors the data and performs the complex calculations needed for the alarm messages

The SQuAl program is shown in Figure 1 (a snapshot taken from the *Aurora* GUI): The input port (1) shows where tuples enter *Aurora* from the outside data source. In this case, it is the application's C++ program that reads in the sensor log file. A Union box (2) serves merely to split the stream into two identical streams. A Map box (3) eliminates all tuple fields except those related to water quality. Each Superbox (4) calculates the sliding window statistics for one of the water quality attributes. The parallel paths (5) form a binary join network that brings the results of (4)'s sub-networks back into a single stream. The top branch in (6) has all the tuples where the fish act oddly, and the bottom branch has the tuples where the fish act normally. For each of the tuples sent into (1) describing abnormal fish behavior, (6) emits an alarm message tuple. This output tuple has the sliding window water quality statistics for both the moment the fish acted oddly, and for the most recent previous moment that the fish acted normally. Finally the output port (7) shows where result tuples are made available to the C++-based monitoring application.

Overall, the entire application consisted of 3400 lines of C++ code (primarily for file-parsing and a simple monitoring GUI) and a 53-operator SQuAl program.

Technical Details

This section provides a more detailed explanation of the application's C++ and SQuAl programs. The casual reader may want to skip ahead to the *Lessons Learned* section below.

Input Data File

The input data file provided by the Army was broken down into reports, each of which described a 15 minute period of readings taken from the fish sensors and the water quality sensors. With 1908 paragraphs, this file described just under 20 days of activity. Those 1908 reports appear in chronological order, and there is no gap in the reports.

This file was produced by specialized software used by the Army lab. High frequency data, sampled many times per second from the fish sensors, was fed to the Army's software. This software detected events such as fish "coughing" or dying, and reported those events in the per-15 minute appearing in the file they provided to us.

We did discuss with the Army lab the possibility of having *Aurora* perform that high-resolution analysis of the sensor data. We eventually agreed ignore this domain for the time being, because they deemed the alarm-sounding problem to be more pressing.

Because the data file provided by the Army lab wasn't specifically tailored for use by the *Aurora* project, it contained some data in each 15 minute report that were superfluous to our application. We won't discuss those data in this paper.

The interesting data for each 15 minute report are as follows:

- A timestamp describing which 15 minute period the report covers.
- A serial number. The reports are numbered 1, 2, 3, ..., 1908.
- For *each* of the eight fish monitored:
  - **Ventilations per minute**. This is simply the fish's average breathing rate during the sample period.
  - **Volts**. This is the average received signal strength of the sensors affixed to the fish's gills.
  - **Coughs per minute**. Fish can cough, and do so under certain hostile condition. This is the average number of coughs per minute during the sample period.
  - **Percent body movement**. A measure of how much the fish's whole body moved during the sample period. A low value indicates a sedentary fish.
- For the entire fish group, a count was made of *how many* fish exhibited unusual values in any of the four measured listed above, during the sample period. An unusual value is referred to as Out of Control (*OOC*). Those group-wide measurements are:
  - Number of fish with out-of-control *Ventilations-per-minute* values.
  - Number of fish with out-of-control *Volts* values.
  - Number of fish with out-of-control *Coughs per minute values.*
  - Number of fish with out-of-control *Percent body movement* values.
  - Number of dead fish. This is the number of fish reckoned to be dead during the sample period.
- **Simple water characteristics**. While summaries of the fish' status were calculated every 15 minutes, the traditional water quality measurements were only taken every 30 minutes. The effect in the data file is that the water quality was reported every 15 minutes, but the values of those measurements would only change in the file every 30 minutes. As stated earlier, the measured water quality properties are pH, oxygenation level, conductivity, and temperature.

C++ Input Processing

Aurora provides a simple C++ library for sending data items into a SQuAl program, and for receiving the results. This program essentially converts the "stream" of reports present in the data file, into a stream of *tuples*, one tuple per report, that's sent into *Aurora* to be processed by the application's SQuAl program.

The Need for Latches

Implementing this application made us aware of a design limitation in the SQuAl language. Recall from our original requirements that when the fish are acting strangely, we need to report not only the present water quality, but also the water quality as of the last time the fish were behaving normally.

For instance, suppose the data file described the following history:

Time	*Fish Behavior*
10:15 a.m.	Normal
10:30 a.m.	Abnormal.
10:45 a.m.	Abnormal.
11:00 a.m.	Abnormal.

Then our application is expected to produce three alerts: One each at 10:30, 10:45, and 11:00. Furthermore, each of those alerts needs to include the details of the water quality at 10:15, the last time the fish behaved normally.

SQuAl didn't provide any functionality for retaining and accessing the values of particular tuples that had appeared earlier in the data stream. For expediency we chose to make these calculations in the C++ program that fed our SQuAl program, rather than rushing a change to the SQuAl language. Since we wrote this application, however, SQuAl has improved to provide this needed "latch" functionality.

Detecting Alarm-worthy Reports

The logic for deciding whether a particular 15-minute report is worth of raising an alarm for can easily be expressed in either a C++ program or a SQuAl program.

Unfortunately the problem mentioned immediately above, in the section "The Need for Latches", complication this part of our application.

As explained above, we needed the C++ program to discover, for each alarm-worthy report, what the water quality was as of the last-non-alarm report. This in turn means the C++ program needed to know whether a report was alarm-worthy or not.

If we had chosen to implement the alarm-worthiness logic in SQuAl then we would have had each report undergo the following data flow:

1. C++ program: Reads the data file and submits its reports to *Aurora*.
2. SQuAl program: Sets a field in each report, indicating whether or not its alarm-worthy.
3. C++ program: If the report is non-alarm-worthy, record its water quality and timestamp for future use. If the report *is* alarm-worthy, store in to the report tuple the water quality and timestamp of the last *non*-alarm-worthy report.
4. SQuAl program: Performs the rest of its logic (sliding average calculations, etc.) on the report tuple.

Aurora wasn't originally designed to permit user-supplied C++ code to perform supplemental processing in the middle of a SQuAl program, as would have occurred in Step 3 above. Therefore, our application performed Steps 1-3 in its C++ program before submitting the the report to SQuAl, even though SQuAl is fully qualified to perform Step 2.

As of this writing, *Aurora* and SQuAl have been improved in two ways either of which would have let us avoid implementing Step 2 in our C++ program:

- SQuAl's "latch" mechanism (offered by its new Read and Update operators), would have permitted Step 3 to be implemented in SQuAl.
- *Aurora* now permits user-defined C++ functions to be readily invoked by a SQuAl program. So even if the application insisted on implementing Step 3 in C++, he could have offered that C++ code as a User Defined Function (UDF).

Box-by-box walkthrough

In this section we'll explore the role each box in the SQuAl program plays in the overall application. The numbering of the subsections below corresponds to the yellow-background labels placed on Figure 1.

Section 1: Input box

The box labeled *i0,* on the left side of the SQuAl program shown in Figure 1, represents where the tuples supplied by the C++ program enter the SQuAl program. This box itself does no work – it just indicates an entry point into the SQuAl program.

The tuples submitted by the C++ program into this box contain the following fields, whose meanings are explained above.

- serial number (integer)

- report timestamp (timestamp)
- water temperature Celsius (floating point)
- pH (floating point)
- conductivity (mS/cm) (floating point)
- dissolved oxygen (mg/l) (floating point)
- Alarm-worthy ("Y" or "N"). Calculated by the C++ program.
  We needed the per-fish measures (ventilations / minute, volts, cough rate, and percent body movement) to calculate this value, but we don't need those per-fish values in subsequent logic. That's why those per-fish data don't appear in this tuple.
- Serial number of the most recent previous report that wasn't alarm-worthy (integer). Calculated by the C++ program.
- Number of fish with out-of-control *Ventilations-per-minute* values (integer)
- Number of fish with out-of-control *Volts* values (integer)
- Number of fish with out-of-control *Coughs per minute values* (integer)
- Number of fish with out-of-control *Percent body movement* values (integer)
- Number of dead fish (integer)

Section 2: Union box

In general, SQuAl permits many arrows to leave a particular box. When a tuple is emitted by the box, a copy of it is placed onto each of those arrows.

A design oversight in the original *Aurora* implementation forced at most *one* arrow to leave an Input box.

Tuples simply pass though a Union box that has only one input arrow. Since multiple arrows can be drawn leaving a a Union box, we used on where to give us the ability to send two copies of the tuples entering the SQuAl program out to different parts of the program. (Newer versions of *Aurora* are free from this problem.)

Section 3: Map box

This box eliminates some of the data fields from the report tuples entering it, leaving only:

- Report serial number
- Report timestamp
- Water quality measurements: pH, conductivity, oxygenation level, and temperature.

At the time this was done because we believed that downstream Aggregate operators required the absence of all irrelevant fields. As of this writing, however, we no longer believe that to be true.

Section 4: Superboxes

In this section we see four Superboxes. Each Superbox is a visual substitute for a collection of boxes and arrows that are present there in the SQuAl network. Using a Superboxes in a SQuAl program is analogous to using a subroutines in a procedural programming language. We used Superboxes here to merely to reduce the visual complexity of the SQuAl program.

Each one of these Superboxes is use to calculate the 1-hour, 2-hour, and 4-hour sliding window change for a particular water quality measure.

For example, each time a tuple enters the top Superbox in section "4", a tuple is emitted that with the following information:

- Serial number of the tuple that just entered the box
- Timestamp of the tuple that just entered the box
- Three fields: The numerical differences between the water temperature of the tuple that just entered the box, and the tuple pertaining to water quality samples taken one, two, and four hours ago.

(Briefly, an Aggregate box performs ongoing calculations, such as "average" or "sum" on some field in the tuples that have recently passed through the box. The Aggregate operator has complicated functionality, and the interested reader is referred to recent papers describing the SQuAl programming language.)

Each one of these Superboxes uses three Aggregate boxes: one Aggregate box for each of the different time lengths over which the deltas were to be computed.

For example with the Water Temperature Superbox: One of its constituent Aggregate boxes emits the 1-hour temperature delta, one Aggregate box emits the 2-hour temperature delta, and one emits the 4-hour temperature delta. Inside each Superboxes, several Join boxes are used to merge the results of the three Aggregate boxes into a single tuple. It's these tuples that are emitted by the overall Superbox.

Section 5: Binary Join network

Each of the Superboxes from section "4" emits a tuple with various statistics about the same water quality sample. In this section, we merge together all of the tuples pertaining to a particular water quality sample into a single tuple. This merge is done ultimately to permit the SQuAl program to emit one tuple bearing all of the information relevant to an alarm, rather than spreading that information over multiple output tuples.

A Join box takes corresponding tuples from its two different input streams, and produces a tuple with the combined fields from those input tuples. In section "4", this "correspondence" is defined as the two input tuples having identical serial numbers.

Because a Join box can take only two input streams, we can't merge the results of all the Superboxes with just one Join box. Instead, we use a binary tree of Join boxes to ultimately produce a single tuple for each input report tuple.

Notice that a Map box follows each Join box. A Map box can be used produce tuples that lack or rename fields that appeared in the box's input stream. Here we use Map boxes to simplify the names of fields (automatically) produced by the Join boxes. As of this writing, the Join operator has been improved so that it doesn't produce output tuples whose fields have hard-to-read names.

Section 6: Reuniting report tuples with their water-quality statistics

Recall that section "5" of the SQuAl program ultimately produces one tuple for each 15-minute report in the Army-supplied data file. These tuples leaving section "5" contain the 1-,2-, and 4-hour changes in pH, conductivity, oxygenation level, and temperature of the water as of the 15-minute period described by that report tuple.

The other stream of tuples coming into section "6" is straight from the Union box in section "2". Unlike the tuples from section "6", the tuples from section "2" continue still contain information such as:

- the *present* value of each of the water quality measures
- the number of fish that had out-of-control ventilation rates, voltages, cough rates, or percent body movements, or were dead
- whether or not the report is alarm-worthy
- the serial number of the most recent previous report that wasn't alarm-worthy

In this section of the SQuAl program, we combine these data with the statistics provide by the tuples leaving section "5" to produce the tuples that ultimately are emitted by this SQuAl program.

The leftmost box in this section is a Filter box. Any report tuples that is alarm-worthy are emitted on the top arrow leaving the Filter box. The report tuples that *aren't* alarm-worthy are emitted on the bottom arrow leaving the filter box.

On each of those two branches, a report tuples, regardless of whether or not it's alarm-worthy, it mated with the tuple from section "5" that bears the water quality statistics calculated for that report. As occurred in section "5", the tuples are mated with each other by the Join box when they have identical values in their Serial Number field.

Finally, the rightmost Join box brings these top and bottom branches together. Recall that report tuples not only have a field giving their own serial number, but also the serial number of the most recent non-alarm-worthy report. This Join box uses these two fields, to produce a tuple with the following information:

- For the alarm-worth tuple: All of the present water/fish data available, as well as the water quality statistics calculated in section "5".
- The same information, but for last non-alarm report tuple preceding the alarm-worthy report.

As we did in sections "4" and "5", we conclude with a Map box to rename awkwardly named fields produced by the preceding Join box.

Section 7: Output box

Just as the Input box of section "1" is a placeholder, this Output box is a do-nothing placeholder representing where the data calculated by this SQuAl program leaves *Aurora* and is made available to the C++ program.

Lessons Learned

During the development of the application, we observed that *Aurora*'s stream model proved very convenient for describing the required sliding-window calculations. For example, a single instance of the aggregate operator computed the 4-hour sliding-window deltas of water temperature.

Aurora's GUI for designing query networks also proved invaluable. As the query network grew large in the number of operators used, there was great potential for overwhelming complexity. The ability to manually place the operators and arcs on a workspace, however, permitted a visual representation of ``subroutine" boundaries that let us comprehend the entire query network as we refined it.

We found that small changes in the SQuAl language design would have greatly reduced our processing network complexity. For example, Aggregate boxes apply some window function (such DELTA(water-pH)), to the tuples a sliding window. Had an Aggregate box been capable of evaluating multiple window functions at the same time on its window (such as DELTA(water-pH) *and* DELTA(water-temp)), we could have used far fewer boxes. Many of these changes have since been made to SQuAl.

The ease with which the processing flow could be experimentally reconfigured during development, while remaining comprehensible, was surprising. It appears that this was only possible by having both a well-suited operator set, and a GUI tool that let us visualize the processing. It seems likely that this application was developed at least as quickly in *Aurora* as it would have been with standard procedural programming. One large benefit the *Aurora* implementation has over a procedural implementation, however, is that we find it easier to comprehend and explain the processing logic by looking at a query network GUI than by reading pages of procedural source code.